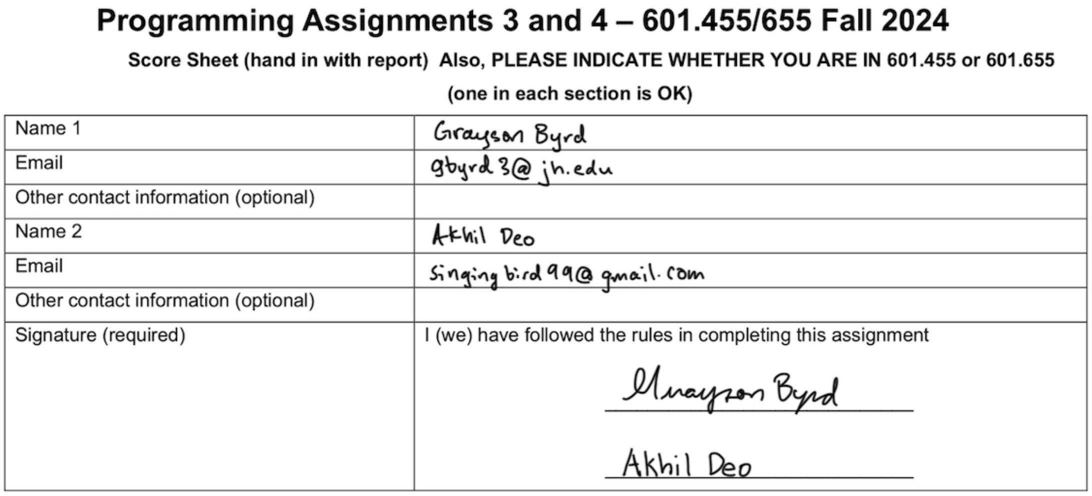
CIS PA3

Akhil Deo and Grayson Bryd

November 2024



Programming Assignments 3 and 4 – 601.455/655 Fall 2024

Score Sheet (hand in with report) Also, PLEASE INDICATE WHETHER YOU ARE IN 601.455 or 601.655

(one in each section is OK)

Name 1	Grayson Byrd
Email	gbyrd3@jh.edu
Other contact information (optional)	
Name 2	Akhil Deo
Email	singingbird99@gmail.com
Other contact information (optional)	
Signature (required)	I (we) have followed the rules in completing this assignment Grayson Byrd Akhil Deo

Figure 1: "I did not cheat" signature.

1 Introduction

In computer integrated surgery, computing the registration between point clouds and 3D meshes is a crucial part of building robust robotic systems. Previously, we implemented a point cloud to point cloud registration algorithm for computing such a registration, however, our prior algorithms are effective only for cases in which the correspondences between the point clouds are known. For many scenarios, this is an unrealistic constraint. For situations where the correspondence is unknown or there is not an exact correspondence between points, a more robust algorithm needs to be developed to address this.

In this programming assignment, we seek to build the matching portion of the Iterative Closest Point algorithm for computing registrations between point clouds with no known correspondence. To do this, we are provided data that describes a system consisting of 1) two rigid bodies, a pointer (with local coordinate frame F_A and a rigid attachment to a bone in the system (with local coordinate frame F_B), measured in the tracker frame via LED optical sensors, and 2) a mesh of a "bone" measured in CT coordinates. Given the measurements for both rigid bodies in local body coordinates, we must first compute the registration between each rigid body and the optical tracking frame. Next, given the location $\vec{A}_{tip}$ of the pointer tip in rigid body coordinates, we must use our computed registration between the rigid bodies and the tracking system to compute the location of $\vec{A}_{tip}$ in the other rigid body coordinate frame. Finally, we can compute the location of $\vec{A}_{tip}$ in the CT coordinate frame by using the registration between F_B and the CT coordinate frame. This registration is denoted as F_{reg} and is assumed to be the identity matrix, I, for this assignment for simplicity. Finally, we will use the point matching algorithm that we develop in this assignment to match the location of the pointer tip that is touching the rigid bone to the closest point on the mesh of the bone measured in CT coordinates. We will use the debug datasets to test and validate our algorithm as well as custom validation approaches that will be discussed later.

2 Description of Mathematical Approach

2.1 Point Cloud Registration

The high level overview of our approach to the 3D point cloud to point cloud registration can be described by the following: given two point cloud sets of equal size with known correspondences, find the optimal transform, $F_{opt} = [R_{opt}, \vec{p}_{opt}]$, that minimized the mean squared error between the point cloud correspondences in the two sets. our approach was taken directly from [1].

I start with a pair point clouds of size N, p_i and p'_i where $i = 1, ..., N$. Each point cloud is measured in a different coordinate system and every ith point in one point cloud corresponds to the ith point in the other point cloud. we transform both point clouds to local coordinate frames centered at the centroid of the point cloud. To do this, first calculate the centroid of the point cloud, $p_{centroid}$.

$$p_{centroid} = (1/N) \sum_{i=1}^{N} p_i \tag{1}$$

Next, use the below equation transform the point cloud to the local coordinate frame centered at the centroid.

$$q_i = p_i - p_{centroid} \tag{2}$$

Where q_i is the ith point in the point cloud in the local coordinate frame centered at the point cloud's centroid.

Now that both point clouds are in the same coordinate frame, we can find the optimal rotation for the transformation between them by solving the following equation:

$$\arg\min_{R} \sum_{i=1}^{N} \|q'_i - Rq_i\|^2 \tag{3}$$

To solve this, we can use Singular Value Decomposition (SVD) as outlined in [1]. Begin by calculating the covariance matrix, H:

$$H = \sum_{i=1}^{N} q_i q_i'^T \tag{4}$$

Next, find the SVD of H:

$$H = USV^T \tag{5}$$

You can then calculate:

$$X = VU^T \tag{6}$$

Then, you calculate the determinant of X. If $\det(X) == +1$, then $R = X$. If $\det(X) == -1$, then the algorithm fails. In this case, we simply flip the third column of V, i.e. $V' = [v_1, v_2, -v_3]$ and then re-compute X as $X = V'U^T$. Then, we set $R = X$.

Finally, to get the translation, t, for the transformation, we simply use the following equation:

$$t = p'_{centroid} - Rp_{centroid} \tag{7}$$

2.2 Calculation of s_k Values

In this subsection, we present the mathematical formulation underpinning the calculation of the s_k sample values. We refer to [4] for this formulation.

The position d_k represents the position of the pointer tip with respect to the rigid body B. This can be mathematically represented as:

$$\vec{d}_k = \mathbf{F}_{B,k}^{-1} \cdot \mathbf{F}_{A,k} \cdot \vec{A}_{\text{tip}} \tag{8}$$

However, we are concerned with the position of the sample points s_k, not d_k. The formulation of s_k can be thought of as:

$$\vec{s}_k = \mathbf{F}_{Reg}^{-1} \cdot \mathbf{d}_k \tag{9}$$

However, since $\mathbf{F}_{Reg}$ is stated to be the identity matrix $\mathbf{I}$, this simplifies to:

$$\vec{s}_k = \mathbf{d}_k \tag{10}$$

$\mathbf{F}_{Reg}$ is the registration between the poses $\mathbf{F}_{B,k}$ and $\mathbf{F}_{CT}$.

2.3 Computation of the Nearest Point on a Triangle

To determine the closest point on a triangle to a given point, we consider a triangle defined by its vertices $\vec{a}$, $\vec{b}$, and $\vec{c}$, and a point $\vec{p}$ in space. The objective is to find the point $\vec{q}$ on the triangle $\triangle abc$ that minimizes the Euclidean distance to $\vec{p}$.

First, we define the edge vectors and the vectors from vertex $\vec{a}$ to point $\vec{p}$:

$$\vec{ab} = \vec{b} - \vec{a}, \quad \vec{ac} = \vec{c} - \vec{a}, \quad \vec{ap} = \vec{p} - \vec{a} \tag{11}$$

We compute the following dot products to determine the relative position of $\vec{p}$ with respect to the triangle's main features (i.e vertices and edges):

$$d_1 = \vec{ab} \cdot \vec{ap} \tag{12}$$
$$d_2 = \vec{ac} \cdot \vec{ap} \tag{13}$$
$$d_3 = \vec{ab} \cdot (\vec{p} - \vec{b}) \tag{14}$$
$$d_4 = \vec{ac} \cdot (\vec{p} - \vec{b}) \tag{15}$$
$$d_5 = \vec{ab} \cdot (\vec{p} - \vec{c}) \tag{16}$$
$$d_6 = \vec{ac} \cdot (\vec{p} - \vec{c}) \tag{17}$$

Now, we can use the dot products to evaluate the position of $\vec{p}$ relative to the triangle and determine the closest point $\vec{q}$ through the following cases.

The first case to consider is if $\vec{p}$ is closest to a vertex of the triangle. In order to determine if this is true, we use the following conditions. Please note that ϵ refers to a small threshold value used for numerical precision. It might be easier to think of ϵ as 0 when initially trying to understand this mathematical approach.

- If $d_1 \leq \epsilon$ and $d_2 \leq \epsilon$, then $\vec{a}$ is the closest point.
- If $d_3 \geq -\epsilon$ and $d_4 \leq d_3$, then $\vec{b}$ is the closest point.
- If $d_6 \geq \epsilon$ and $d_5 \leq d_6$, then $\vec{c}$ is the closest point.

The second case to consider is if the point $\vec{p}$ is closest to one of the edges $\vec{ab}$, $\vec{ac}$, or $\vec{bc}$. Within that, consider the following:

- If $v_c = d_1 d_4 - d_3 d_2 \leq \epsilon$, $d_1 \geq \epsilon$, and $d_3 \leq \epsilon$, the closest point lies on edge $\boldsymbol{\vec{ab}}$:

$$v = \frac{d_1}{d_1 - d_3}, \quad \vec{q} = \vec{a} + v\vec{ab} \tag{18}$$

- If $v_b = d_5 d_2 - d_1 d_6 \leq \epsilon$, $d_2 \geq \epsilon$, and $d_6 \leq \epsilon$, the closest point lies on edge $\boldsymbol{\vec{ac}}$:

$$w = \frac{d_2}{d_2 - d_6}, \quad \vec{q} = \vec{a} + w\vec{ac} \tag{19}$$

- If $v_a = d_3 d_6 - d_5 d_4 \leq \epsilon$, $(d_4 - d_3) \geq \epsilon$, and $(d_5 - d_6) \geq \epsilon$, the closest point lies on edge $\boldsymbol{\vec{bc}}$:

$$w = \frac{d_4 - d_3}{(d_4 - d_3) + (d_5 - d_6)}, \quad \vec{q} = \vec{b} + w(\vec{c} - \vec{b}) \tag{20}$$

If none of the above conditions are satisfied, the closest point lies within the face of the triangle. We compute the barycentric coordinates v and w as follows:

$$\text{denom} = \frac{1}{v_a + v_b + v_c} \tag{21}$$

$$v = \frac{v_b}{\text{denom}} \tag{22}$$

$$w = \frac{v_c}{\text{denom}} \tag{23}$$

The closest point is then given by:

$$\vec{q} = \vec{a} + v\vec{ab} + w\vec{ac} \tag{24}$$

This piecewise case-by-case approach ensures that the closest point $\vec{q}$ is accurately determined, whether it lies on a vertex, an edge, or within the interior of the triangle. For this approach, we referred to Dr. Taylor's slides [7]

2.4 Naive Search

In our naive search, for each point in the point cloud, we compute the closest point to each triangle in the mesh and take the point on the mesh with the smallest distance. This is a naive but inefficient approach and runs in O(n) (linear) time. This algorithm was described in the PA3 assignment file [4].

2.5 Efficient Search Using k-d Tree

To efficiently determine the closest point on a mesh to a given point $\vec{p}$, we employ a k-d tree data structure. A k-d tree is a hierarchical tree-based data structure that partitions based on spatial coordinates. This allows for rapid nearest neighbor searches, which making it ideal for handling large and complex meshes. [2]. For all of the following, we refer to Dr. Taylor's slides [7].

2.5.1 k-d Tree Construction

The first step involves constructing a k-d tree from the centroids of the mesh's triangles. Given a mesh represented by its vertices $\mathbf{V}$ and a list of triangle indices $\mathbf{i}$, the centroid of each triangle is calculated as:

$$\text{centroid}_i = \frac{1}{3}(\vec{v}_{a_i} + \vec{v}_{b_i} + \vec{v}_{c_i}) \tag{25}$$

where $\vec{v}_{a_i}$, $\vec{v}_{b_i}$, and $\vec{v}_{c_i}$ are the vertices of the i-th triangle.

Then, the centroids are organized into a k-d tree, which recursively partitions the 3D space along alternating axes (x, y, z). The construction process is the following:

1. Initialization: Start with all triangle centroids and set the initial depth to 0.
2. Axis Selection: At each node, select the splitting axis based on the current depth (axis = depth%3).
3. Median Computation: Sort the points along the selected axis and choose the median point. This ensures a balanced tree.
4. Recursive Partitioning: Split the points into left and right sets, based on the median, and then build the left and right subtrees in a recursive manner.

Mathematically, the median point along the chosen axis divides the set of points into two nearly equal halves, optimizing the search efficiency.

2.5.2 Nearest Neighbor Search

To find the closest point on the mesh to a query point $\vec{p}$:

1. k Nearest Centroids: Query the k-d tree to retrieve the k nearest triangle centroids to $\vec{p}$. This reduces the search space by focusing only on the most relevant triangles. The algorithm is as follows.
   (a) To query the tree, you treat is as a binary search tree where the search value determining whether you go right or left in the tree is the i axis of the point where i = depth % D where D is the dimension of the point.
   (b) As you descend the tree, searching in the fashion mentioned above, you keep track of the minimum distance and the point of each node in the search space.
   (c) As each depth in the search effectively cuts the parent subspace into two smaller subspaces, once you reach a leaf node you will recurse back up the tree, calculating the minimum distance to the search space provided by the alternate path which was not originally taken. If the distance to a search space is less than the current minimum distance to a node point, you will also search to a leaf node of that entire subspace following the procedure in 2.
   (d) Using this search method, we achieve an average time complexity of O(logn), which is a significant speedup over the O(n) time complexity naive approach.
2. Closest Point Evaluation: For each of the k retrieved triangles, compute the closest point on the triangle to $\vec{p}$ using the method described in Section 2.3. Use the point with the minimum Euclidean distance.

3 Description of Algorithmic Approach

Below is psuedocode for the algorithmic implementation of the mathematical approaches outlined in Section 2. We implemented the code in Pyton 3.11. Below, we have broken each major function into its own pseudocode algorithmic implmentation. In our algorithms, we make extensive use of the NumPy [3], PyTest [5], and SciPy [6] libraries.

3.1 Utility Functions

Algorithm 1 Find Closest Point on Mesh

```
Input: p (query point), v (vertex array), triangles (list of triangle vertex indices)
Output: closest_point (closest point on mesh), min_dist (minimum distance to mesh)
min_dist ← inf
closest_point ← None
for triangle_indices in triangles do
    a ← v[triangle_indices[0]]
    b ← v[triangle_indices[1]]
    c ← v[triangle_indices[2]]
    cp ← closest_point_on_triangle(p, a, b, c)
    distance ← norm(p - cp)
    if distance < min_dist then
        min_dist ← distance
        closest_point ← cp
    end if
end for
return closest_point, min_dist
```

Algorithm 2 Find Closest Point on Mesh (Slow Version)

```
Input: p (query point), mesh (dictionary containing vertex array V and triangle indices i)
Output: closest_point (closest point on mesh), min_dist (minimum distance to mesh)
closest_point, min_dist ← find_closest_point(p, mesh['V'], mesh['i'])
return closest_point, min_dist
```

Algorithm 3 Find Closest Point on Mesh (Fast Version using KDTree)

```
Input: p (query point), mesh (dictionary with vertex array V and triangle indices i),
kdtree (KDTree of triangle centroids), triangle_indices_list (list of triangle vertex indices),
num_neighbors (number of nearest neighbors to query, default = 5)
Output: closest_point (closest point on mesh), min_dist (minimum distance to mesh)
v ← mesh['V']
distances, indices ← kdtree.query(p, k=num_neighbors)
if num_neighbors = 1 then
    indices ← [indices]
else
    indices ← atleast_1d(indices)
end if
triangles ← [triangle_indices_list[idx] for idx in indices]
closest_point, min_dist ← find_closest_point(p, v, triangles)
return closest_point, min_dist
```

Algorithm 4 Build KDTree of Triangle Centroids

```
Input: mesh (dictionary containing vertex array V and triangle indices i)
Output: kdtree (KDTree built from triangle centroids), centroids (array of centroids),
triangle_indices_list (list of tuples defining triangles)
v ← mesh['V']
i ← mesh['i']
centroids ← empty list
triangle_indices_list ← empty list
for idx, triangle_indices in enumerate(i) do
    a ← v[triangle_indices[0]]
    b ← v[triangle_indices[1]]
    c ← v[triangle_indices[2]]
    centroid ← (a + b + c) / 3.0
    centroids.append(centroid)
    triangle_indices_list.append(triangle_indices)
end for
centroids ← np.array(centroids)
kdtree ← KDTree(centroids)
return kdtree, centroids, triangle_indices_list
```

Algorithm 5 Find Closest Point on Triangle

```
Input: p (query point), a (vertex 1 of the triangle), b (vertex 2 of the triangle), c (vertex 3 of the triangle)
Output: closest_point (closest point on the triangle to p)
e ← small tolerance value (e.g., $10^{-8}$)
if p is close to a, b, or c (within e) then
    return the closest of a, b, or c
end if
ab ← b - a
ac ← c - a
ap ← p - a
d1 ← dot(ab, ap)
d2 ← dot(ac, ap)
if d1 ≤ e and d2 ≤ e then
    return a
end if
bp ← p - b
d3 ← dot(ab, bp)
d4 ← dot(ac, bp)
if d3 ≥ 0 and d4 ≤ d3 then
    return b
end if
vc ← d1 * d4 - d3 * d2
if vc ≤ e and d1 ≥ e and d3 ≤ e then
    v ← d1 / (d1 - d3)
    return a + v * ab
end if
cp ← p - c
d5 ← dot(ab, cp)
d6 ← dot(ac, cp)
if d6 ≥ e and d5 ≤ d6 then
    return c
end if
vb ← d5 * d2 - d1 * d6
if vb ≤ e and d2 ≥ e and d6 ≤ e then
    w ← d2 / (d2 - d6)
    return a + w * ac
end if
va ← d3 * d6 - d5 * d4
if va ≤ e and d4 - d3 ≥ e and d5 - d6 ≥ e then
    w ← (d4 - d3) / ((d4 - d3) + (d5 - d6))
    return b + w * (c - b)
end if
denom ← 1.0 / (va + vb + vc)
v ← vb * denom
w ← vc * denom
return a + ab * v + ac * w
```

3.2 Main Functions

These functions are high level functions that are found in the *main.py* file. They utilize the lower level *Utility* functions to solve specific problems like the ones outlined in this assignment. This promotes re-usability of the code, as our utility functions can be used over and over again in the future while our main functions can implement assignment specific code that will only be used once for very specific applications.

Algorithm 6 Compute d_k

```
Input: F_A (transformation matrix for body A), F_B (transformation matrix for body B), A_tip (tip of body A as a 3D point)
Output: d_k (position of the pointer tip with respect to the rigid body B)
F_B_inv ← F_B.inverse()
A_tip_homogeneous ← A_tip.reshape(1, 3)
A_tip_tracker ← F_A.transform_pts(A_tip_homogeneous)[0]
d_k ← F_B_inv.transform_pts(A_tip_tracker.reshape(1, 3))[0]
return d_k
```

Algorithm 7 Process Frame and Compute Values

```
Input: k (frame index), sample_readings (list of sample readings), body_a (data for body A), body_b (data for body B), mesh (mesh data), kdtree (KDTree for closest point search), triangle_indices_list (list of triangle indices)
Output: c_k, s_k, distance_k, elapsed_slow, elapsed_fast
A_markers_tracker ← sample_readings[k]["A"]
B_markers_tracker ← sample_readings[k]["B"]
A_markers_body ← body_a["Y"]
B_markers_body ← body_b["Y"]
F_A_k ← pcd_to_pcd_reg_w_known_correspondence(A_markers_body, A_markers_tracker)
F_B_k ← pcd_to_pcd_reg_w_known_correspondence(B_markers_body, B_markers_tracker)
d_k ← compute_d_k(F_A_k, F_B_k, body_a["t"])
s_k ← d_k                                        ▷ Since F_reg = I, so s_k = d_k
start_time_slow ← current time
c_k_slow, distance_k_slow ← closest_point_on_mesh_slow(s_k, mesh)
end_time_slow ← current time
elapsed_slow ← end_time_slow - start_time_slow
start_time_fast ← current time
c_k_fast, distance_k_fast ← closest_point_on_mesh_fast(s_k, mesh, kdtree, triangle_indices_list, num_neighbors=10)
end_time_fast ← current time
elapsed_fast ← end_time_fast - start_time_fast
difference ← norm(c_k_slow - c_k_fast)
if not close_enough(c_k_slow, c_k_fast, tolerance=1e-6) then
    print warning with frame index k and difference
end if
return c_k_slow, s_k, distance_k_slow, elapsed_slow, elapsed_fast
```

Algorithm 8 Main Script for Programming Assignment #3

```
Input: dataset_prefix (prefix for the dataset, e.g., "PA3-A-Debug-")
Retrieve body_a, body_b, mesh, sample_readings, and num using retrieve_data(dataset_prefix)
num_frames ← length of sample_readings
Build KDTree: kdtree, centroids, triangle_indices_list ← build_triangle_centroid_kdtree(mesh)
slow_time ← 0.0
fast_time ← 0.0
data ← empty list
for k in range(num_frames) do
    c_k_slow, s_k, distance_k_slow, elapsed_slow, elapsed_fast ← process_frame(k, sample_readings, body_a, body_b, mesh, kdtree, triangle_indices_list)
    slow_time ← slow_time + elapsed_slow
    fast_time ← fast_time + elapsed_fast
    Append [c_k_slow[0], c_k_slow[1], c_k_slow[2], s_k[0], s_k[1], s_k[2], distance_k_slow] to data
end for
Print performance improvements using print_performance_improvements(slow_time, fast_time, dataset_prefix)
Calculate and output MSE using calculate_and_output_mse(data, dataset_prefix)
Write output to file using save_output(dataset_prefix, num_frames, data, num)
```

4 Overview of Program Structure

```
main_PA3.py
|
|-- get_file_paths(dataset_prefix)
|
|-- retrieve_data(dataset_prefix)
|
|-- compute_d_k(F_A, F_B, A_tip)
|
|-- process_frame(k, sample_readings, body_a, body_b, mesh, kdtree, triangle_indices)
|   |-- pcd_to_pcd_reg_w_known_correspondence(A_markers_body, A_markers_tracker)
|   |     [utils.pcd_2_pcd_reg]
|   |-- pcd_to_pcd_reg_w_known_correspondence(B_markers_body, B_markers_tracker)
|   |     [utils.pcd_2_pcd_reg]
|   |-- compute_d_k(F_A_k, F_B_k, body_a["t"])
|   |-- closest_point_on_mesh_slow(s_k, mesh)
|   |   |-- find_closest_point(p, mesh['V'], mesh['i'])              [utils.closest_point]
|   |-- closest_point_on_mesh_fastest(s_k, mesh, kdtree, triangle_indices_list, num_neighbors)
|   |   |-- find_closest_point(p, mesh['V'], mesh['i'])              [utils.closest_point]
|
|-- calculate_and_output_mse(data, dataset_prefix)
```

```
|   |-- parse_output(output_file_path) [utils.data_processing]
|
|-- print_performance_improvements(slow_time, fast_time, dataset_prefix)
|
|-- main(dataset_prefix)
|   |-- retrieve_data(dataset_prefix)
|   |-- build_triangle_centroi_kdtree(mesh)
|   |   |-- KDTree (class)                                     [utils.kdtree]
|   |   |   |-- KDTreeNode (class)                             [utils.kdtree]
|   |-- process_frame(k, sample_readings, body_a, body_b, mesh, kdtree, triangle_indices_list
|   |-- print_performance_improvements(slow_time, fast_time, dataset_prefix)
|   |-- calculate_and_output_mse(data, dataset_prefix)
|   |-- save_output(dataset_prefix, num_frames, dataset, num)
|
|-- full_run()
|   |-- For each dataset prefix ("pa2-debug-a-" for example):
|       |-- main(prefix)
```

4.1 Description of Code Files

File Path from PROGRAMS Dir	Description
main_pa3.py	Functions for completing PA33 specific problems.
utils/data_processing.py	Functions for parsing the datasets.
utils/pcd_2_pcd_reg.py	Functions for 3D point cloud to 3D point cloud registration.
utils/pivot_cal.py	Functions for performing a pivot calibration.
utils/transform.py	Custom FT class used to perform frame transformation on 3D points.
utils/interpolation.py	Functions for conducting distortion correction
tests/test_utils.py	Basic helper functions for use in the test functions.
utils/closest_point.py	Functions for finding the closest point on a mesh (fast and slow methods)
utils/kdtree.py	Implements the KDTree class
utils/kdtree_node.py	Implements the KDTree node class
tests/test_closest_point.py	Testing and validation of closest point algos with custom datasets.

Table 1: Description of each code file

4.2 Reusability and Modularity

We made our code modular and reused it constantly throughout this project in many instances. The code found in the PROGRAMS/utils/data_processing.py file provides functions to parse the various datasets in the DATA/ directory. Instead of re-writing the parsing whenever we needed it, we wrote the function once, tested it, and re-used it. We did the same thing with our point cloud registration function, using it consistently throughout the PA3 workflow, as well as with the closest_point_on_triangle function in closest_point.py. We also made sure to break down the workflow into a set of concise and modular functions. This reduces the tech debt in our code, as well as makes it easier to read and understand. These functions are described in the Algorithmic Approach section.

Additionally, we wrote several small helper scripts that were too trivial/extraneous to include in the algorithmic approach section, which we've reused constantly throughout the codebase. These are shown in Section 4.3.

4.3 Description of helper functions

Function Name	Description
get_file_paths	Takes dataset_prefix and returns the true file path of that file
retrieve_data	Takes dataset_prefix and returns the data of the dataset
save_output	Takes dataset prefix, p_{tip}^{CT} and saves formatted to output2.txt file
print_performance_improvements	Prints the performance gains between fast and slow matching method

Table 2: Description of each helper functions

5 Discussion of Validation Approach

In order to validate our approach we did two kinds of testing: comparison of our final predicted values to that of the debug datasets and custom creation of data to validate our slow and fast closest point matching algorithms.

5.1 Generating Synthetic Data

Generating synthetic data for a closest point matching algorithm between a 3D point and a mesh is not an easy task. There are quite a few edge cases that need to be addressed, so a complicated algorithm was required to achieve a unit test that made us confident about our approach. Since the debug datasets have some added noise, we will not achieve a MSE of 0 when comparing our predicted values to the ground truth values provided in the debug dataset answers. Therefore, in order to sanity check our approach, we decided to generate ground truth data with no noise. When doing this, we can validate our approach by ensuring that our predicted values are **exactly** the same as the ground truth values. In this way we can feel even more confident about our approach since we expect to see some error between our predicted values and the debug dataset answers, although we do no know how much error.

To create a custom validation dataset for 3D point matching to the closest point laying on a 3D mesh, we first needed an algorithm to generate a random 3D mesh. An important constraint for generating this mesh is that it needed to be convex. This convexity contraint is what allows us to generate the ground truth data for the closest point on the mesh as well as the distance from the 3D point to the closest point on the mesh. The pseudocode for this function is provided in Algorithm 9.

Algorithm 9 Generate Random Convex Polygon

```
1: procedure GENERATE_RANDOM_CONVEX_POLYGON($N$)
2:     Generate $N$ random points in 3D space and store in points
3:     Compute the convex hull of points and store in hull
4:     points $\leftarrow$ vertices used in hull
5:     hull $\leftarrow$ convex hull of updated points
6:     return hull
7: end procedure
```

Algorithm 10 Generate Random Point on Triangle Plane

```
1: procedure RANDOM_POINT_ON_TRIANGLE_PLANE(vertices, triangle_indices_row)
2:     Let $A$, $B$, $C$ $\leftarrow$ vertices specified by triangle_indices_row
3:     Generate two random values $u$ and $v$ from a uniform distribution in $[0, 1]$
4:     if $u + v > 1$ then
5:         Set $u \leftarrow 1 - u$ and $v \leftarrow 1 - v$        ▷ Reflect values to ensure point is within triangle
6:     end if
7:     return $(1 - u - v) \cdot A + u \cdot B + v \cdot C$ ▷ Barycentric coordinates for a point on the triangle
8: end procedure
```

Algorithm 11 Generate Random Point on Triangle Edge

```
1: procedure RANDOM_POINT_ON_TRIANGLE_EDGE(vertices, triangle_indices_row)
2:     Let $A$, $B$, $C$ $\leftarrow$ vertices specified by triangle_indices_row
3:     Define edges as the list of edges: $\{(A, B), (B, C), (C, A)\}$
4:     Select a random edge from edges and assign to edge
5:     Generate a random value $t$ from a uniform distribution in $[0, 1]$
6:     return $(1 - t) \cdot$ edge$[0] + t \cdot$ edge$[1]$                ▷ Point along the chosen edge
7: end procedure
```

Algorithm 12 Get Normal Unit Vector from Triangle

```
procedure GET_NORMAL_UNIT_VECTOR_FROM_TRIANGLE(vertices, triangle_indices, convex_hull)
    Let $a$, $b$, $c \leftarrow$ vertices specified by triangle_indices
    Compute edge vectors: ab $\leftarrow b - a$ and ac $\leftarrow c - a$
    Compute the cross product normal_vector $\leftarrow$ cross(ab, ac)
    Compute the norm of normal_vector and store as norm
    if norm is close to 0 then
        Raise error: ”The triangle vertices are collinear; normal vector cannot be defined.”
    end if
    Define two unit normal vectors: unit_vec_1 $\leftarrow$ normal_vector/norm and unit_vec_2 $\leftarrow$ −unit_vec_1
    Set offset_distance to a small value, e.g., 5
    Calculate test points: test_point_1 $\leftarrow a +$ unit_vec_1 $\cdot$ offset_distance and test_point_2 $\leftarrow a +$ unit_vec_2 $\cdot$ offset_distance
    Check if test_point_1 is inside the hull using is_point_inside_hull and store result in test_point_1_inside
    Check if test_point_2 is inside the hull using is_point_inside_hull and store result in test_point_2_inside
    if test_point_1_inside and test_point_2_inside then
        Assert failure: at least one test point should be outside the hull
    end if
    if test_point_1_inside equals test_point_2_inside then
        return None    ▷ Both points are either inside or outside, so direction is undetermined
    end if
    if test_point_1_inside then
        return unit_vec_2    ▷ Return unit vector pointing outside
    else
        return unit_vec_1    ▷ Return unit vector pointing outside
    end if
end procedure
```

Algorithm 13 Generate Test Closest Point Test Case

```
1: procedure GENERATE_TEST_CLOSEST_POINT_TEST_CASE(num_vertices)
2:     convex_hull ← Generate a random convex hull using generate_random_convex_polygon(num_vertices)
3:     vertices ← Points of convex_hull restricted to its vertices
4:     triangle_indices ← Triangles (simplices) of convex_hull
5:     Initialize empty lists test_pcd, nearest_points, and dist
6:     for each triangle t in triangle_indices do
7:         pt_on_plane ← Generate a random point on the plane of t using random_point_on_triangle_plane(vertices, t)
8:         Append pt_on_plane to test_pcd and nearest_points
9:         Append 0 to dist
10:        pt_on_edge ← Generate a random point on an edge of t using random_point_on_triangle_edge(vertices, t)
11:        Append pt_on_edge to test_pcd and nearest_points
12:        Append 0 to dist
13:        vertex_idx ← Random integer between 0 and 2
14:        pt_on_vertex ← Vertex at vertex_idx of t in vertices
15:        Append pt_on_vertex to test_pcd and nearest_points
16:        Append 0 to dist
17:        norm_unit_vec ← Normal unit vector pointing outside the convex hull from t using get_normal_unit_vector_from_triangle(vertices, t, convex_hull)
18:        if norm_unit_vec is not None then
19:            distance ← Random float between 0 and 10
20:            Append pt_on_plane + norm_unit_vec · distance to test_pcd
21:            Append pt_on_plane to nearest_points and distance to dist
22:            distance ← Random float between 0 and 10
23:            Append pt_on_edge + norm_unit_vec · distance to test_pcd
24:            Append pt_on_edge to nearest_points and distance to dist
25:            distance ← Random float between 0 and 10
26:            Append pt_on_vertex + norm_unit_vec · distance to test_pcd
27:            Append pt_on_vertex to nearest_points and distance to dist
28:        end if
29:    end for
30:    mesh ← Dictionary with keys V (set to vertices) and i (set to triangle_indices)
31:    return test_pcd, mesh, nearest_points, dist
32: end procedure
```

Algorithm 14 Test Closest Point Algorithm (Slow)

```
 1: procedure TEST_CLOSEST_POINT_ALGORITHM_SLOW
 2:     test_pcd, mesh, nearest_points, distances ← Generate test data using generate_test_closest_point_test_case(1500)
 3:     Initialize empty lists pred_closest and pred_dists
 4:     for each point p in test_pcd do
 5:         closest, dist ← Find the closest point and distance on the mesh for p using closest_point_on_mesh_slow(p, mesh)
 6:         Append closest to pred_closest and [dist] to pred_dists
 7:     end for
 8:     Convert pred_closest and pred_dists to arrays
 9:     Assert that the sum of squared differences between pred_closest and nearest_points, and between pred_dists and distances, is close to zero:

            np.isclose(np.average((pred_closest - nearest_points)**2), 0)

10: end procedure
```

Algorithm 15 Test Closest Point Algorithm (Fast)

```
 1: procedure TEST_CLOSEST_POINT_ALGORITHM_FAST
 2:     test_pcd, mesh, nearest_points, distances ← Generate test data using generate_test_closest_point_test_case(1500)
 3:     Initialize empty lists pred_closest and pred_dists
 4:     kdtree, centroids, triangle_indices_list ← Build KD-tree for triangle centroids using build_triangle_centroid_kdtree(mesh)
 5:     for each point p in test_pcd do
 6:         closest, dist ← Find the closest point and distance on the mesh for p using closest_point_on_mesh_fast(p, mesh, kdtree, triangle_indices_list, num_neighbors=100)
 7:         Append closest to pred_closest and [dist] to pred_dists
 8:     end for
 9:     Convert pred_closest and pred_dists to arrays
10:     Assert that the sum of squared differences between pred_closest and nearest_points, and between pred_dists and distances, is close to zero:

            np.isclose(np.average((pred_closest - nearest_points)**2), 0)

11: end procedure
```

From the convex hull object, you can then get the vertices and triangle indices of the mesh in the same format that is used in PA3. Next, we use the inherent properties of a convex polygon to generate ground truth data for our validation approach. For our ground truth data, we create the following arrays:

1. test_pcd: the point cloud of points we want to match to the closest point on the mesh

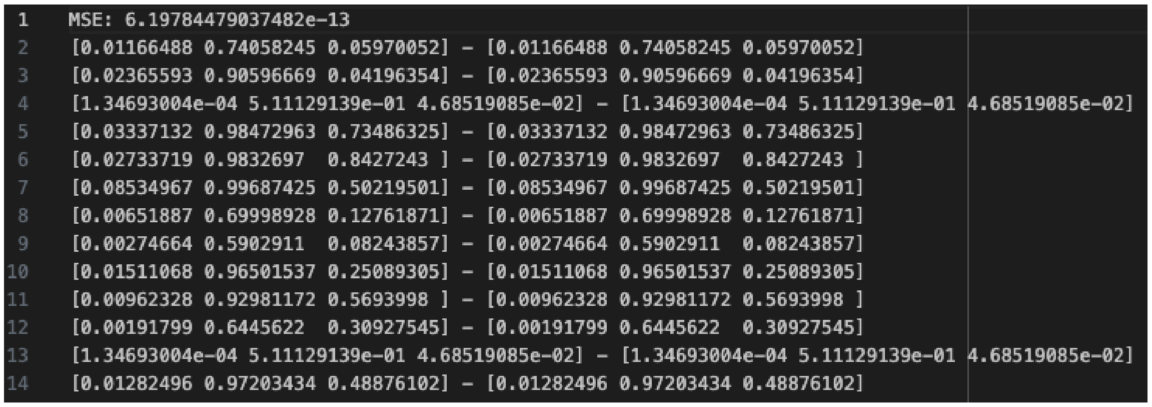
2. nearest_points: the nearest point sitting on the mesh corresponding to the test point in test_pcd with the same index
3. distances: the distances from the test point to the nearest point on the mesh corresponding to the test point and nearest point at the same index

To create out synthetic dataset for each triangle in the mesh, we do the following:

1. **Add a point to the test points that falls on the plane formed by the triangle.** To do this, we simply get a random point that lies on the triangle formed by the three vertices. We do this using Algorithm 10. We then add this random point on the triangle as the test point and also the nearest point and set our distance to 0.
2. **Add a point to the test points that falls on one of the edges of the triangle.** To do this, we simply get a random point that lies on one of the edges of the triangle using Algorithm 11. We then add this random point on the triangle as the test point and also the nearest point and set our distance to 0.
3. **Add a point to the test points that is one of the vertices of the triangle.** We randomly select one of the vertices of the triangle and add this point to the test points and nearest points and set the distance to 0.
4. **Add a point to the test points whose nearest point is on the triangle, but the actual test point itself is not on the mesh.** To do this we take advantage of the convexity constraint of the mesh. First, we find a random point on the triangle like in Case 1. Next, we find the normal vector that points out of the convex polygon. We must find the vector that points out of the polygon because if the vector points inside the polygon, we will not know what the nearest point is. If the normal vector points out of the polygon, we know that our starting randomly chosen point on the triangle is our nearest point on the mesh. This unit vector is found using Algorithm 12. Next, we randomly scale the unit normal vector and add the scaled unit normal vector the the initially chosen random point on the mesh. Finally, we add this point as our test point, add the initial randomly selected point as our nearest point, and add the scale that was applied to the normal unit vector to our distance.
5. **Add a point to the test points whose nearest point is on one of the edges of the triangle, but whose actual point is not on the edge of the triangle.** To do this, we followed essentially the same procedure as in 4 but the initial point was randomly selected from one of the edges of the triangle.
6. **Add a point to the test points whose nearest point is on one of the vertices of the triangle, but whose actual point is not on a vertex of the triangle.** Once again, the procedure for doing this follows 4, except the initial point is randomly chosen from the vertices of the triangle.

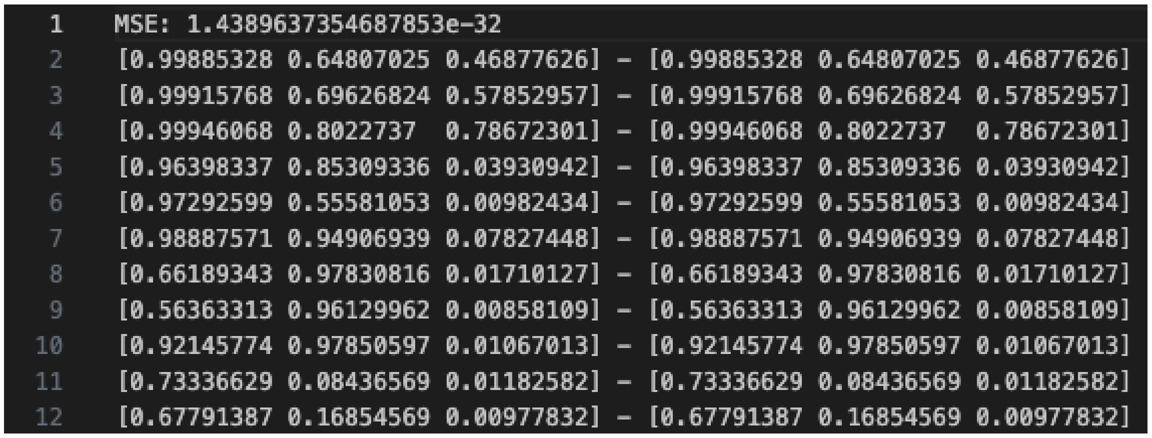
The final procedure for generating our test data is shown in pseudocode in Algorithm 13. Finally, to validate our approach, we compare our predicted values using our slow and fast algorithms for matching the closest point on the mesh to the ground truth data. We use the PyTest [5] framework and assert that our predicted values are identical to our ground truth values. If you run "pytest" in the terminal when inside our repository, you will see that both these tests pass. The pseudocode for these tests is shown in Algorithms 14 and 15.

Additionally, we save out the predicted and true values for each method as well as the MSE between the predicted and ground truth values for the fast and the slow method. The Figures 2 and 3 show the MSE for the slow and fast methods as well as the predicted values on the left separated by a dash and the ground truth values on the right of the dash. Note that the MSE is essentially zero and the error is negligible. Notice also that the values of the predicted and ground truth are identical as well. This shows that our approach for closest point matching is validated. The discrepancy between the points for the slow vs fast method is because we generated a different random convex polygon for each, but the results still show that both methods are sound.



```
MSE: 6.19784479037482e-13
 [0.01166488 0.74058245 0.05970052] - [0.01166488 0.74058245 0.05970052]
 [0.02365593 0.90596669 0.04196354] - [0.02365593 0.90596669 0.04196354]
 [1.34693004e-04 5.11129139e-01 4.68519085e-02] - [1.34693004e-04 5.11129139e-01 4.68519085e-02]
 [0.03337132 0.98472963 0.73486325] - [0.03337132 0.98472963 0.73486325]
 [0.02733719 0.9832697  0.8427243 ] - [0.02733719 0.9832697  0.8427243 ]
 [0.08534967 0.99687425 0.50219501] - [0.08534967 0.99687425 0.50219501]
 [0.00651887 0.69998928 0.12761871] - [0.00651887 0.69998928 0.12761871]
 [0.00274664 0.5902911  0.08243857] - [0.00274664 0.5902911  0.08243857]
 [0.01511068 0.96501537 0.25089305] - [0.01511068 0.96501537 0.25089305]
 [0.00962328 0.92981172 0.5693998 ] - [0.00962328 0.92981172 0.5693998 ]
 [0.00191799 0.6445622  0.30927545] - [0.00191799 0.6445622  0.30927545]
 [1.34693004e-04 5.11129139e-01 4.68519085e-02] - [1.34693004e-04 5.11129139e-01 4.68519085e-02]
 [0.01282496 0.97203434 0.48876102] - [0.01282496 0.97203434 0.48876102]
```

Figure 2: Custom dataset validation results for the Slow method.



```
MSE: 1.4389637354687853e-32
 [0.99885328 0.64807025 0.46877626] - [0.99885328 0.64807025 0.46877626]
 [0.99915768 0.69626824 0.57852957] - [0.99915768 0.69626824 0.57852957]
 [0.99946068 0.8022737  0.78672301] - [0.99946068 0.8022737  0.78672301]
 [0.96398337 0.85309336 0.03930942] - [0.96398337 0.85309336 0.03930942]
 [0.97292599 0.55581053 0.00982434] - [0.97292599 0.55581053 0.00982434]
 [0.98887571 0.94906939 0.07827448] - [0.98887571 0.94906939 0.07827448]
 [0.66189343 0.97830816 0.01710127] - [0.66189343 0.97830816 0.01710127]
 [0.56363313 0.96129962 0.00858109] - [0.56363313 0.96129962 0.00858109]
 [0.92145774 0.97850597 0.01067013] - [0.92145774 0.97850597 0.01067013]
 [0.73336629 0.08436569 0.01182582] - [0.73336629 0.08436569 0.01182582]
 [0.67791387 0.16854569 0.00977832] - [0.67791387 0.16854569 0.00977832]
```

Figure 3: Custom dataset validation results for the Fast method.

5.2 Validation with Debug Datasets

In order to validate our approach and ensure that our approach is sound when using it blindly on the "Unkown" datasets, we first utilized the debug datasets to determine if our approach seemed sound.

To do this, we loaded in all of the data from each debug dataset, and calculated the predicted $\vec{c}_k$ values. We then compared these values to debug dataset by taking the mean squared error between the predicted and expected values. The values for mean squared error is shown in Table 4.

6 Discussion of Results

To determine the accuracy of our work, we calculated the Mean Squared Error (MSE) between each coordinate in the 3D points output in the predicted output file (<dataset_prefix>Output) and in the debug datasets. The table below shows the results of this error calculation for each of the debug data sets. For all the debug datasets, the mean squared error is quite small (<1.05), indicating that our results are highly accurate. The slight discrepancy between our predicted results and the target output is likely due to noise somewhere in the system, potentially in the tracker measurements of the LED optical markers on the rigid bodies. This would make our registration slightly off and could affect our final predictions as the position of the pointer tip in relation to the CT frame would have slight errors in it.

Name	Slow Time (sec)	Fast Time (sec)	Speedup Multiple	Mean Square Error
pa3-debug-a	0.92704	0.00497	186.61408x	0.00002
pa3-debug-b	0.96098	0.00538	178.61553x	0.90276
pa3-debug-c	1.16587	0.00533	218.54834x	0.32020
pa3-debug-d	0.94056	0.00550	171.15602x	1.00353
pa3-debug-e	0.90344	0.00498	181.48958x	1.02359
pa3-debug-f	0.91442	0.00504	181.41819x	0.60874

Table 3: Error analysis

The results of our error analysis and validation give us confidence in our overall approach, and although we cannot see the correct answers of the unknown dataset, through exhaustive debugging and rigorous testing we are confident that our results are accurate. Below is the statistical summary of the distance data $\vec{d_k}$ - $\vec{c_k}$, as well as all tabulated results from the unknown datasets.

Name	Minimum Distance	Maximum Distance	Average Distance	Standard Deviation
pa3-debug-a	0.00021	0.00801	0.00327	0.00241
pa3-debug-b	0.02120	3.64111	1.47135	0.99371
pa3-debug-c	0.02567	2.34952	0.81093	0.67958
pa3-debug-d	0.08297	4.71956	1.40771	1.23731
pa3-debug-e	0.12426	3.97318	1.57975	1.04492
pa3-debug-f	0.08231	2.69708	1.28553	0.69235
pa3-unknown-g	0.09455	5.02468	2.06043	1.57513
pa3-unknown-h	0.04681	2.74124	1.34192	0.84911
pa3-unknown-j	0.04681	2.74124	1.34192	0.84911

Table 4: Statistical Summary of Each Dataset (Across Its Frames of Data)

$\vec{d_{k,x}}$	$\vec{d_{k,y}}$	$\vec{d_{k,z}}$	$\vec{c_{k,x}}$	$\vec{d_{k,y}}$	$\vec{d_{k,z}}$	$\parallel \vec{d_k}$ - $\vec{c_k} \parallel$
-33.90	-23.80	-13.72	-34.02	-23.94	-13.62	0.208
12.35	21.25	-18.23	11.42	21.55	-18.27	0.976
20.94	3.27	-27.19	18.09	1.80	-28.63	3.518
19.80	15.99	47.99	15.70	13.11	47.58	5.025
-37.89	-12.40	-11.21	-37.69	-12.33	-11.41	0.293
4.87	15.11	-5.39	3.60	16.70	-6.86	2.506
-5.06	-2.14	48.94	-9.31	-4.57	48.82	4.898
10.95	-6.49	51.29	11.29	-7.89	51.20	1.442
33.33	-6.59	-17.54	32.80	-6.21	-17.40	0.658
17.19	18.81	-30.61	17.08	18.79	-30.34	0.297
35.89	-3.06	-13.95	35.26	-2.79	-13.99	0.683
-3.80	-17.89	-39.15	-3.86	-17.85	-39.09	0.095
-36.05	-5.41	-42.77	-38.46	-4.45	-45.51	3.775
-28.95	9.43	-31.88	-29.13	10.03	-32.14	0.676
6.72	-9.43	16.25	6.97	-10.32	16.48	0.957
31.21	17.73	-18.86	29.01	16.51	-18.03	2.651
15.59	-16.38	-12.80	14.75	-19.24	-13.90	3.173
19.58	24.59	10.80	18.60	21.27	10.05	3.542
-12.22	4.10	13.14	-14.81	3.90	13.51	2.616
18.43	-7.98	-23.32	16.82	-8.43	-26.08	3.219

Table 5: Data from PA3-G-Unknown-Output.txt

$\vec{d_{k,x}}$	$\vec{d_{k,y}}$	$\vec{d_{k,z}}$	$\vec{c_{k,x}}$	$\vec{d_{k,y}}$	$\vec{d_{k,z}}$	$\parallel \vec{d_k}$ - $\vec{c_k} \parallel$
34.20	-5.66	-16.99	36.25	-7.33	-17.71	2.741
7.31	-15.82	-16.41	8.36	-17.95	-17.08	2.467
1.18	18.44	4.58	2.19	17.83	5.38	1.423
39.12	1.58	3.88	40.28	1.02	3.85	1.294
15.56	21.72	39.36	15.74	22.17	39.48	0.494
8.43	-13.57	-19.18	9.46	-15.40	-20.63	2.558
39.73	6.10	-6.71	39.69	6.09	-6.69	0.047
-39.72	-8.64	-39.38	-38.00	-8.69	-38.92	1.786
28.70	17.99	-22.85	28.63	17.94	-22.79	0.103
-6.48	2.07	-36.79	-7.58	1.13	-36.19	1.566
-0.61	-0.60	63.40	-0.62	-0.60	62.82	0.582
20.95	12.48	54.90	22.08	12.87	54.96	1.203
-4.93	15.75	9.43	-4.45	15.17	9.92	0.897
-29.52	-11.67	-47.41	-29.41	-11.63	-47.03	0.395
18.42	17.24	51.33	19.42	18.29	51.46	1.455
-26.62	2.56	-43.67	-25.96	0.82	-41.81	2.624
-10.82	-14.75	-46.34	-10.65	-14.87	-46.62	0.351
-4.24	3.83	63.22	-4.03	4.00	61.81	1.439
-40.21	-13.89	-14.79	-41.04	-14.03	-14.22	1.027
17.40	-17.71	-9.67	17.16	-20.04	-10.15	2.386

Table 6: Data from PA3-H-Unknown-Output.txt

$\vec{d_{k,x}}$	$\vec{d_{k,y}}$	$\vec{d_{k,z}}$	$\vec{c_{k,x}}$	$\vec{d_{k,y}}$	$\vec{d_{k,z}}$	$\parallel \vec{d_k}$ - $\vec{c_k} \parallel$
0.76	-0.16	-24.67	1.28	-0.11	-25.50	0.974
21.76	-5.79	18.70	22.61	-8.38	18.98	2.745
27.53	2.21	25.31	29.21	1.58	25.92	1.902
35.57	-3.94	-14.93	38.98	-5.39	-14.71	3.706
14.90	-16.58	-3.27	15.06	-19.12	-2.43	2.679
18.75	9.19	63.06	18.69	9.21	62.28	0.779
14.55	-9.82	9.74	15.24	-12.00	10.47	2.396
-21.39	-22.37	-45.80	-22.08	-23.79	-47.66	2.439
27.62	1.40	-30.03	27.56	1.31	-30.21	0.211
8.34	-6.98	28.69	8.80	-9.85	28.89	2.911
0.82	17.93	4.56	1.70	16.88	5.34	1.584
22.79	-2.31	32.25	24.77	-4.26	32.79	2.840
30.66	-8.04	3.70	31.99	-10.20	3.87	2.547
-25.67	-16.35	-47.74	-25.95	-16.47	-48.81	1.112
28.47	-2.44	18.16	29.96	-3.86	19.07	2.247
-10.83	-19.02	-45.23	-9.75	-18.94	-48.23	3.182
-32.56	-25.77	-38.18	-33.14	-26.53	-38.76	1.111
1.77	22.51	9.88	3.04	20.59	10.36	2.357
-8.85	-29.10	-25.70	-6.14	-32.65	-25.05	4.508
-7.99	1.03	27.71	-6.71	1.22	27.44	1.320

Table 7: Data from PA3-J-Unknown-Output.txt

7 Who Did What?

Akhil worked on the basic implementation of the point to closest point on 3D mesh matching algorithm while Grayson worked on the validation approach. The writeup was split evenly between partners with Grayson taking the Introduction, Discussion of Validation approach, Algorithmic Approach, Program Structure and Conclusion, and Akhil taking the Description of Mathematical Approach. We are both happy with the amount of work each member of the team did.

References

[1] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(5):698–700, 1987.

[2] k-d tree. https://en.wikipedia.org/wiki/K-d_tree.

[3] NumPy, Numerical Python. http://www.numpy.org.

[4] Programming assignments 3 and 4 – 601.455/655 fall 2024. https://ciis.lcsr.jhu.edu/lib/exe/fetch.php?media=courses:455-655:2024:programming_3_and_4_600-445-2024.pdf.

[5] PyTest, Python Testing. https://docs.pytest.org/en/stable/.

[6] SciPy, Scientific Python. http://www.scipy.org.

[7] Russell H. Taylor. 600.455/655 lecture notes, 1996-2024.